JavaServer Faces(JSF)

La tecnología JavaServer Faces es un framework de interfaz de componentes de usuarios del lado del servidor para las aplicaciones web basadas en la tecnología Java. Los principales componentes de la tecnología JSF son:

- Una API para:Representar componentes de Interfaz de Usuario (UI) y gestionar su estado.
    - Manejar eventos, validar en el servidor y conversión de datos.
    - Definir la navegación de páginas.
    - Soporte de internacionalización y accesibilidad.
- Dos librerías de etiquetas JSP personalizadas para expresar componentes en una página JSP y enlazar los componentes a objetos del servidor.

El modelo de programación bien definido y las librerías de etiquetas facilitan la construcción y mantenimiento de las aplicaciones web con Interfaces de Usuario (UI) de servidor. Con un mínimo esfuerzo se podría:

- Poner componentes en una página mediante etiquetas de componentes.
- Enlazar eventos generados por componentes con código de la aplicación en el servidor.
- Relacionar componentes UI en una página con datos del servidor.
- Construir una UI con componentes reutilizables y extensibles.
- Salvar y restaurar el estado de la UI más allá de la vida de las peticiones.

JSF es una especificación desarrollada por la Java Community Process. Actualmente existen tres versiones de esta especificación:

- JSF 1.0 (11-03-2004)
- JSF 1.1 (27-05-2004) ## Especificación JSR-127: http://jcp.org/en/jsr/detail?id=127
- JSF 1.2 (11-05-2006) ## Especificación JSR-252: http://jcp.org/en/jsr/detail?id=252

Hoy en día se encuentran disponibles varias implementaciones de JSF, sin embargo la más extendida es Apache MyFaces. ##sta es la implementación más usada. Actualmente implementa la versión 1.2 de la especificación JSF.

Por hacer un resumen, JSF proporciona:

1. Una clara separación entre vista y modelo.
2. Desarrollo basado en componente, no en peticiones.
3. Las acciones del usuario se ligan muy fácilmente al código en el servidor.
4. Creación de familias de componentes visuales para acelerar el desarrollo.
5. Ofrece múltiples posibilidades de elección entre distintos desarrollos.

Esta ficha presenta el contenido acerca de JSF, se muestran a continuación enlaces directos a distintos aspectos:

Características

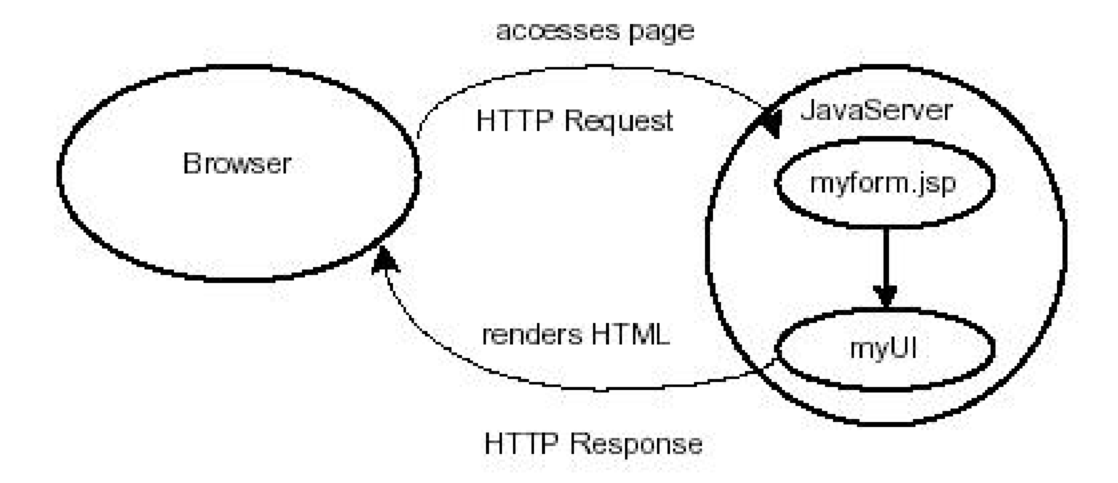
Los principales componentes de la tecnología JavaServer Faces son:

- Un API y una implementación de referencia para: representar componentes UI y manejar su estado; manejo de eventos, validación del lado del servidor y conversión de datos; definir la navegación entre páginas; soportar internacionalización y accesibilidad; y proporcionar extensibilidad para todas estas características.
- Una librería de etiquetas JavaServer Pages (JSP) personalizadas para dibujar componentes UI dentro de una página JSP.

Este modelo de programación bien definido y la librería de etiquetas para componentes UI facilitan de forma significativa la tarea de la construcción y mantenimiento de aplicaciones Web con UIs del lado del servidor. Con un mínimo esfuerzo, podemos:

- Conectar eventos generados en el cliente a código de la aplicación en el lado del servidor.
- Mapear componentes UI a una página de datos del lado del servidor.
- Construir un UI con componentes reutilizables y extensibles.
- Grabar y restaurar el estado del UI más allá de la vida de las peticiones de servidor.

Como se puede apreciar en la siguiente figura, el interface de usuario que creamos con la tecnología JavaServer Faces (representado por myUI en el gráfico) se ejecuta en el servidor y se renderiza en el cliente.



La página JSP, **myform.jsp**, dibuja los componentes del interface de usuario con etiquetas personalizadas definidas por la tecnología JavaServer Faces. El UI de la aplicación Web (representado por myUI en la imagen) maneja los objetos referenciados por la página JSP:

- Los objetos componentes que mapean las etiquetas sobre la página JSP.
- Los oyentes de eventos, validadores, y los conversores que están registrados en los componentes.
- Los objetos del modelo que encapsulan los datos y las funcionalidades de los componentes específicos de la aplicación.

Ventajas e inconvenientes de JSF

Existen numerosas ventajas que hacen que JSF sea una tecnología apropiada para el desarrollo de aplicaciones web:

- Una de las grandes ventajas de la tecnología JavaServer Faces es que ofrece una clara separación entre el comportamiento y la presentación. Las aplicaciones Web construidas con tecnología JSP conseguían parcialmente esta separación. Sin embargo, una aplicación JSP no puede mapear peticiones HTTP al manejo de eventos específicos del componentes o manejar elementos UI como objetos con estado en el servidor.
- La tecnología JavaServer Faces permite construir aplicaciones Web que implementan una separación entre el comportamiento y la presentación tradicionalmente ofrecidas por arquitectura UI del lado del cliente. JSF se hace fácil de usar al aislar al desarrollador del API de Servlet.
- La separación de la lógica de la presentación también le permite a cada miembro del equipo de desarrollo de una aplicación Web enfocarse en su parte del proceso de desarrollo, y proporciona un sencillo modelo de programación para enlazar todas las piezas.
- Otro objetivo importante de la tecnología JavaServer Faces es mejorar los conceptos familiares Disadvantage de componente-UI y capa-Web sin limitar a una tecnología de script particular o un lenguaje de marcas. Aunque la tecnología JavaServer Faces incluye una librería de etiquetas JSP personalizadas para representar componentes en una página JSP, los APIs de la tecnología JavaServer Faces se han creado directamente sobre el API JavaServlet. Esto permite hacer algunas cosas: usar otra tecnología de presentación junto a JSP, crear componentes propios personalizados directamente desde las clases de componentes, y generar salida para diferentes dispositivos cliente. Así, se podrán encapsular otras tecnologías como Ajax en componentes JSF, haciendo su uso más fácil y productivo, al aislar al programador de ellas.
- JavaServer Faces ofrece una gran cantidad de componentes opensource para las funcionalidades que se necesiten. Los componentes Tomahawk de MyFaces y ADFFaces de Oracle son un ejemplo. Además, también existe una gran cantidad de herramientas para el desarrollo IDE en JSF al ser el estándar de JAVA.
- La tecnología JavaServer Faces proporciona una rica arquitectura para manejar el estado de los componentes, procesar los datos, validar la entrada del usuario, y manejar eventos.
- Además, ofrece una rápida adaptación para nuevos desarrolladores.

No obstante, el uso de JavaServer Faces también tiene un conjunto de desventajas:

- Su naturaleza como estándar hace que la evolución de JSF no sea tan rápida como pueda ser la de otros entornos como WebWork, Wicket, Spring , etc.

Distintas implementaciones de JSF

Actualmente existen muchas librerías de etiquetas JSF que pueden complementar a la implementación de la especificación oficial. La elección no tiene porqué cerrarse sobre una de ellas sino que pueden combinarse según interese.

MyFaces Tomahawk

Desarrollado por Apache: http://myfaces.apache.org/tomahawk/ Este conjunto de componentes también es compatible con la implementación de SUN, así como con cualquier implementación compatible con JSF 1.1. Pueden verse los distintos componentes de MyFaces Tomahawk en el siguiente enlace:

- http://www.irian.at/myfaces/

Además pueden consultarse las principales características del proyecto Tomahawk en la siguiente dirección web:

- http://www.marinschek.com/myfaces/tiki/tiki-index.php?page=Features

MyFaces Sandbox

Desarrollado por Apache: http://myfaces.apache.org/sandbox/ Sandbox es un subproyecto de MyFaces que sirve como base de pruebas para las nuevas incorporaciones al proyecto de Tomahawk. Consiste sobre todo en componentes, pero como el proyecto de Tomahawk, puede también contener otras utilidades para JSF. Los distintos componentes pueden consultarse en el siguiente enlace:

- http://www.irian.at/myfaces-sandbox/

ICEfaces

Desarrollado por ICEsoft: http://www.icesoft.com/products/icefaces.html

ICEFaces proporciona un entorno de presentación web para aplicaciones JSF que mejora el framework JSF estándar y el ciclo de vida con características interactivas basadas en AJAX. Para trabajar con ICEfaces puede elegirse cualquiera de las dos implementaciones estándar. En la siguiente dirección web pueden encontrarse demos sobre sus componentes:

- http://www.icesoft.com/products/demos_icefaces.html

RichFaces

Rich Faces es un framework de código abierto que añade capacidad Ajax dentro de aplicaciones JSF existentes sin recurrir a JavaScript. Rich Faces incluye ciclo de vida, validaciones, conversores y la gestión de recursos estáticos y dinámicos. Los componentes de Rich Faces están construidos con soporte Ajax y un alto grado de personalización del ##look-and-feel## que puede ser fácilmente incorporado dentro de las aplicaciones JSF.

- http://labs.jboss.com/jbossrichfaces/

AJAX blueprints components

Desarrollado por java.net: https://blueprints.dev.java.net/ajaxcomponents.html Consiste en una serie de componentes AJAX basados en la tecnología JSF. Tienen dos librerías de componentes JSF. Una basada en la versión 1.2 de JSF y que puede ser usada en un servidor de aplicaciones JEE5. El otro conjunto tiene componentes basados en JSF 1.1 y que pueden ejecutarse en servidores de aplicaciones J2EE 1.4. En la página oficial presentan demos de algunos componentes.

JSF Extensions

Desarrollado por java.net: https://jsf-extensions.dev.java.net/nonav/mvn/ Este proyecto almacena el desarrollo de software para extender las capacidades de la especificación JSF. El software de este proyecto está pensado para que funcione en cualquier implementación que cumple la especificación JSF, aunque según la página oficial las capacidades se mejoran cuando se ejecuta con la implementación de SUN. Este proyecto está dividido en tres, pudiendo trabajar de forma conjunta o por separado. La separación es en función del ciclo de vida del desarrollo: run time, test time y design time.

Ajax4jsf

Fue desarrollado por java.net y patrocinado por Exadel. A partir del 05/03/2007 Exadel y Red Hat decidieron colaborar en el desarrollo de esta y otras tecnologías. La página oficial del proyecto se ha movido a: http://labs.jboss.com/jbossajax4jsf/ Actualmente el proyecto se encuentra dentro del proyecto RichFaces. Ajax4JSF es una extensión opensource para el estándar JSF que añade capacidades AJAX a las aplicaciones JSF sin la necesidad de escribir código Javascript. RichFaces es una librería de componentes para JSF construida sobre Ajax4jsf. Permite una integración fácil de capacidades AJAX en el desarrollo de aplicaciones

de negocio. RichFaces mejora el framework Ajax4jsf de dos formas importantes. Primero, aumenta el número de componentes visuales listos para usar. En segundo lugar, implementa completamente la característica skinnability de incluir un gran número de temas (skins) predefinidos. Con esta característica resulta mucho más fácil gestionar el look-and-feel de una aplicación. Pueden visualizarse algunos ejemplos en la siguiente URL:

- http://livedemo.exadel.com/richfaces-demo/welcome.jsf

RC Faces (Rich Client Faces)

Desarrollado por Vedana: http://www.rcfaces.org/ RC Faces es una librería JSF que proporciona un conjunto de componentes para construir la siguiente generación de aplicaciones web. RC Faces usa tecnologías AJAX y una API de Javascript orientada a objetos para construir páginas de forma dinámica. Es compatible con la implementación estándar de JSF. Desde la página oficial se muestran algunos ejemplos de esta librería:

- http://www.rcfaces.org/starter/index.jsf

ADF Faces

Todos los componentes de ADF Faces han sido donados por Oracle a la Fundación Apache, la cual lo acogió bajo el proyecto Trinidad y ahora se hace el lanzamiento de Apache MyFaces Trinidad Core 1.2.1. Proporciona un amplio conjunto de componentes JSF, siempre permitiendo las interacciones con AJAX, que simplifican radicalmente el desarrollo de aplicaciones web.

Otras implementaciones estudiadas

- Sistemas basados en AJAX-Tags
- Sistemas basados en AjaxAnywhere
- Librería TagLibs Tobago
- Trinidad
- Sistemas basados en Rich Faces

Tabla Comparativa

Por último veamos una tabla comparativa de estas herramientas. Esta tabla suele actualizarse con frecuencia y presenta la comparativa de las últimas librerías de componentes:

- http://www.jsfmatrix.net/

Requisitos e incompatibilidades

En relación con el uso de JSF, se han detectado lo siguientes requisitos agrupados por categoría:

- Versiones de Java.
  1. JDK 1.4.x
  2. JDK 1.5.x
- Contenedor de Servlet.
  1. Tomcat 4.x
  2. Tomcat 5.x
  3. JRun 4 (SP1a)
  4. JBoss 3.2.x
  5. JBoss 4.0.x
  6. BEA Weblogic 8.1

7. Jonas 3.3.6 w/ Tomcat
8. Resin 2.1.x
9. Jetty 4.2.x
10. Jetty 5.1.x
11. Websphere 5.1.2
12. OC4J

No obstante cualquier motor de servlet que cumpla la especificación 2.3 debería valer. Respecto a JSP, con la versión 1.2 es bastante, pero MyFaces usa características de la versión 2.0, por lo tanto en estos contenedores habrá que instalar un jar adicional que viene con la distribución, jsp-2.0.jar.

IMPORTANTE. No añadir ese jar si no es necesario, por ejemplo, en tomcat 5.5, la presencia de este archivo causaría que el motor de servlet dejara de funcionar.

Además habría que tener en cuenta las siguientes consideraciones:

- El motor de servlet debe ser compatible con la especificación 2.3 y las JSP deben ser acordes a la especificación 1.2.
- JSF únicamente soporta peticiones realizadas con POST.
- La especificación no obliga a que haya validaciones en el cliente, si bien dos desarrollos como MyFaces y Shale proporcionan esta posibilidad.

Relación con otros subsistemas

JSF es una especificación y como tal no tiene ninguna incompatibilidad con otros sistemas. Por ello, se pueden escoger los desarrollos que se desean para las necesidades de cada uno, el de jakarta, el de sun , el de oracle, o incluso el de IBM. Habría que estudiar cada uno de estos desarrollos para saber cuáles son las incompatibilidades que cada uno hace constar.

Limitaciones que impone la especificación

- Todo desarrollo de esta especificación tiene que soportar JSP como tecnología de vista. Pero JSF no obliga a que esta sea la opción por defecto. De hecho hay otros proyectos de código abierto que proporcionan la posibilidad de escribir la capa cliente sin usar JSP, como son Facelets y Clay. Concretamente, con Facelets podríamos escribir toda esta capa en HTML.
- El hecho de que JSP sea una tecnología obligatoria, hace que podamos seguir desarrollando con las etiquetas JSP con las que estemos más acostumbrados, como pueda ser JSTL.
- Puede coexistir con Struts en una misma aplicación web. Esto lo hace especialmente importante para las aplicaciones ya escritas en Struts que quieran ir migrando poco a poco a JSF. Se puede cambiar unas pocas páginas y ver los cambios de manera paulatina. Haciendo, por lo tanto, muy suave la transición.

Modo de empleo de JSF

Para el caso de utilización e instalación de JSF se hará uso del proyecto de Apache: MyFaces.
Para ello será necesario descargar las librerías correspondientes de JSF que se encuentran en http://myfaces.apache.org/binary.cgi .

Una vez descargado, se extraen las librerías necesarias incorporándolas dentro de las librerías del nuevo proyecto, quedando una estructura similar a la que se muestra en la figura 1.

- Jakarta Commons BeanUtils. Version 1.6.
- Jakarta Commons Collections. Version 3.0.
- Jakarta Commons Digester. Version 1.5.
- Jakarta Commons Logging. Version 1.0.4.

Los siguientes son obligatorios si se tiene pensado usar componentes tomahawk.

- Jakarta Commons Codec. Version 1.2.
- Jakarta Commons FileUpload. Version 1.0.

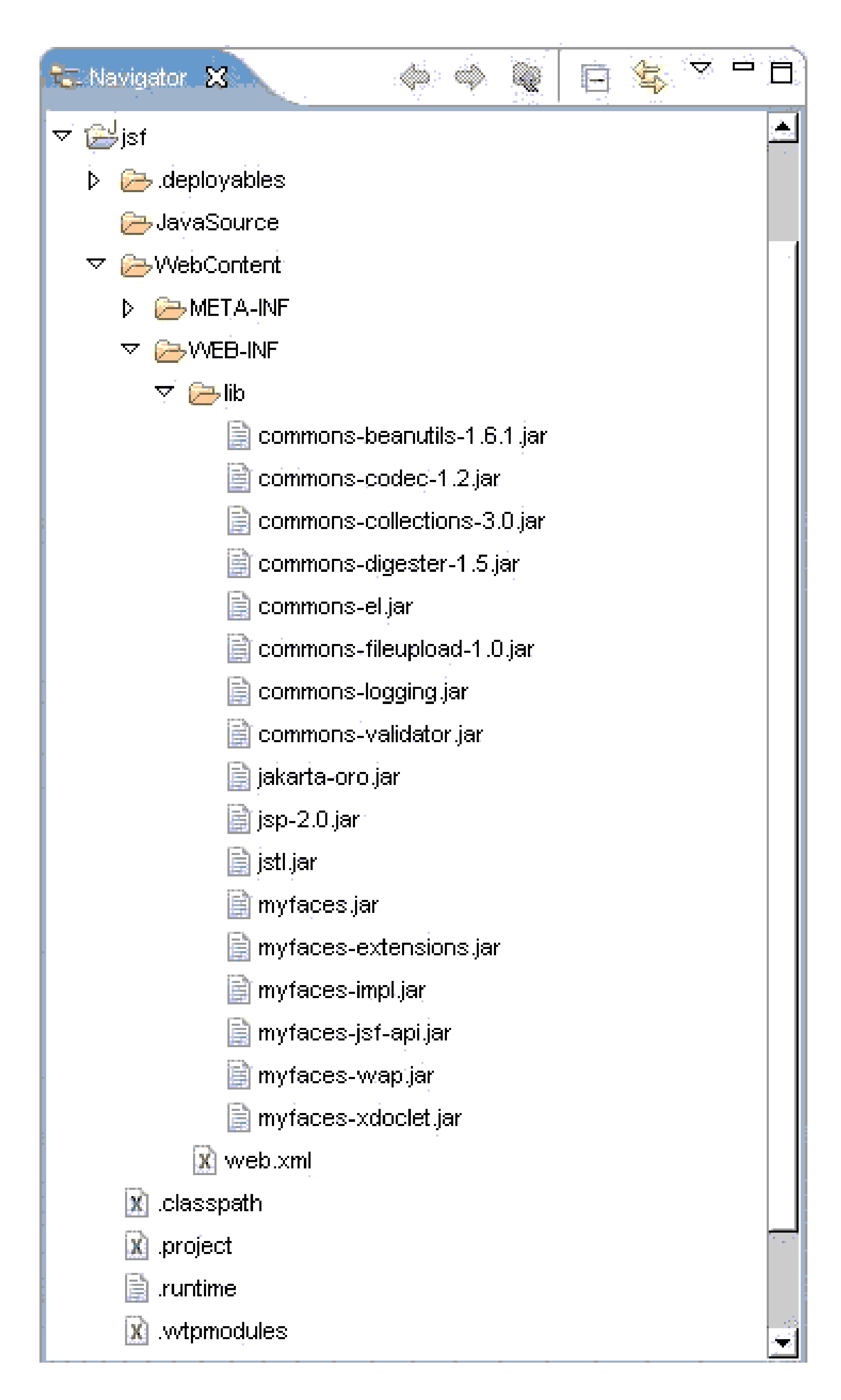

Navigator
jsf
.deployables
JavaSource
WebContent
META-INF
WEB-INF
lib
commons-beanutils-1.6.1.jar
commons-codec-1.2.jar
commons-collections-3.0.jar
commons-digester-1.5.jar
commons-el.jar
commons-fileupload-1.0.jar
commons-logging.jar
commons-validator.jar
jakarta-oro.jar
jsp-2.0.jar
jstl.jar
myfaces.jar
myfaces-extensions.jar
myfaces-impl.jar
myfaces-jsf-api.jar
myfaces-wap.jar
myfaces-xdoclet.jar
web.xml
.classpath
.project
.runtime
.wtpmodules

figura 1

También será necesario modificar el fichero web.xml para que el servlet de JSF recoja todas las peticiones que acaben en ##.jsf##:

```
<?xml version=

"1.0"?>
<web-app id=

"WebApp_ID" version=

"2.4"
 xmlns=

"http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi=

"http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation=

"http://java.sun.com/xml/ns/j2ee
 http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
<context-param>
 <description>
 State saving method:

"client" or

"server" (=

default)
 See JSF Specification 2.5.2
 </description>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
</context-param>
<context-param>
    <description>
    This parameter tells MyFaces

if javascript code should be allowed in the
     rendered HTML output.
    If javascript is allowed, command_link anchors will have javascript code
    that submits the corresponding form.
    javascript is not allowed, the state saving info and nested parameters
    will be added as url parameters.
    Default:

"

true"

    </description>
    <param-name>org.apache.myfaces.ALLOW_JAVASCRIPT</param-name>
    <param-value>

true</param-value>
</context-param>
<context-param>
    <description>
    If

true, rendered HTML code will be formatted, so that it is

"human readable".
    i.e. additional line separators and whitespace will be written, that

do not
    influence the HTML code.
    Default:

"
```

```
true"

    </description>
    <param-name>org.apache.myfaces.PRETTY_HTML</param-name>
    <param-value>

true</param-value>
</context-param>
<context-param>
    <param-name>org.apache.myfaces.DETECT_JAVASCRIPT</param-name>
    <param-value>

false</param-value>
</context-param>
<context-param>
    <description>
     If

true, a javascript function will be rendered that is able to restore the
     former vertical scroll on every request. Convenient feature

if you have pages
     with

long lists and you

do not want the browser page to always jump to the top


if you trigger a link or button action that stays on the same page.
       Default:

"

false"

   </description>
    <param-name>org.apache.myfaces.AUTO_SCROLL</param-name>
    <param-value>

false</param-value>
</context-param>
<!-- Listener, that does all the startup work (configuration, init). -->
<listener>
    <listener-class>org.apache.myfaces.webapp.StartupServletContextListener</listener-class>
</listener>
<!-- Faces Servlet -->
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>Faces Servlet</servlet-name>
   <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
<!-- Welcome files -->
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

En cuanto a las posibles etiquetas jsf disponibles y su funcionalidad puede verse el documento Componentes JSF.pdf que se encuentra en el repositorio.

Enlaces de interés de JSF

- Página oficial de Myfaces
- ADF Faces Components
- Framework AME de Endesa

Recomendaciones de uso

Paso de parámetros a los actions

Con los tags h:commandLink y h:commandButton se puede invocar un método del backing bean utilizando el atributo action o actionListener, pero no se le puede pasar un parámetro directamente por lo que el tag f:attribute puede resultar útil usándolo junto con el actionListener, un ejemplo:

```
<h:form>
    <h:commandLink actionListener=

"#{miBean.action}">
        <f:attribute name=

"nombreAtributo1" value=

"valorAtributo1" />
        <f:attribute name=

"nombreAtributo2" value=

"valorAtributo2" />

        <h:outputText value=

"De click aquí" />
    </h:commandLink>
    <h:commandButton value=

"Click" actionListener=

"#{miBean.action}">
        <f:attribute name=

"nombreAtributo1" value=

"valorAtributo1" />
        <f:attribute name=

"nombreAtributo2" value=

"valorAtributo2" />

    </h:commandButton>
</h:form></pre>
```

Luego, estos atributos pueden ser recuperados utilizando el método getAttributes() del componente que implementa el ActionEvent que manejó el actionListener.

```
package ejemplo;

import javax.faces.event.ActionEvent;

import es.juntadeandalucia.cice.util.FacesUtil;
```

```
public class MyBean {

public void action(ActionEvent event) {

String strAtributo1 = FacesUtil.getActionAttribute(event,

"nombreAtributo1");

String strAtributo2= FacesUtil.getActionAttribute(event,

"nombreAtributo2");
        ...
    }
}
```

```
package es.juntadeandalucia.cice.util;

import javax.faces.event.ActionEvent;

public class FacesUtil {

public

static

String getActionAttribute(ActionEvent event,

String name) {

return (

String) event.getComponent().getAttributes().get(name);
    }
}
```

Por lo que las variables strAtributo1 y strAtributo2 ahora ya tienen los valores asignados de nombreAtributo1 y nombreAtributo2 respectivamente. Se debe tener en cuenta que cada nombre de atributo debe ser único y no sobreescribir atributos por defecto del componente como "id", "name", "value", "binding", "rendered", etc.

Pasar objetos de request en request

Si se tiene un bean con scope de request y se quiere reutilizar una propiedad, parámetro u objeto en la siguiente peticion sin tener que reinicializarlo otra vez y se puede utilizar el tag h:inputhidden para guardar el objeto:

```
<h:form>
    ...
    <h:inputHidden value=

"#{miBean.value}" />
    ...
</h:form>
```

También se puede utilizar un RequestMap para pasar objetos al siguiente request, teniendo en cuenta que serán "limpiados" después de un request.

```
package es.juntadeandalucia.cice.util;

import javax.faces.context.FacesContext;

public class FacesUtil {


public

static

Object getRequestMapValue(FacesContext context,

String key) {


return context.getExternalContext().getRequestMap().get(key);
    }


public

static void setRequestMapValue(FacesContext context,

String key,

Object value) {
        context.getExternalContext().getRequestMap().put(key, value);
    }
}
```

Otra manera es utilizar un SessionMap para mantener los valores que deben ser guardados durante la sesión del usuario:

```
package es.juntadeandalucia.cice.util;

import javax.faces.context.FacesContext;

public class FacesUtil {



public

static

Object getSessionMapValue(FacesContext context,

String key) {


return context.getExternalContext().getSessionMap().get(key);
    }


public

static void setSessionMapValue(FacesContext context,

String key,

Object value) {
        context.getExternalContext().getSessionMap().put(key, value);
    }


public
```

```
static

Object removeSessionMapValue(FacesContext context,

String key) {


return context.getExternalContext().getSessionMap().remove(key);
    }
}
```

Comunicación entre ManagedBeans

Es posible tener más de un managed bean en un scope, si es absolutamente necesario por diseño, entonces se puede utilizar el método getSessionMap() de FacesContext para comunicar los beans durante una sesión de navegador.
Un ejemplo de dos managed beans declarados en el fichero faces-config.xml:

```
<managed-bean>
    <managed-bean-name>miBean1</managed-bean-name>
    <managed-bean-class>ejemplo.MiRequestBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
<managed-bean>
    <managed-bean-name>miBean2</managed-bean-name>
    <managed-bean-class>ejemplo.MiSessionBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Tanto miBean1 como miBean2 serán accesibles desde cualquier página JSF, no importando que el scope de ambos sea distinto. El scope de miBean1 es request por lo que en cada petición se creará una nueva instancia mientras que el de miBean2 está puesto en session, en cuyo caso la misma instancia del bean será utilizada durante toda la sesión.

Con la finalidad de obtener y asignar los valores en el SessionMap, podría ser útil crear una superclase con algunos métodos protected que fueran heredados por cada backing bean, o sólo poner el mapa en una clase utility como la que se está viendo en los ejemplos, FacesUtil, de cualquier manera sería:

```
FacesContext.getCurrentInstance().getExternalContext().getSessionMap().get(key);
FacesContext.getCurrentInstance().getExternalContext().getSessionMap().put(key, value);
```

Uso de datatable

Las tablas dinámicas que se pueden mostrar con el tag h:dataTable reciben un objeto de tipo List o DataModel con una colección de beans o pojos, o un mapa de objetos que los contiene.

Por ejemplo para una tabla que contenga tres campos: ID, Nombre y Valor. Se crearía una clase wrapper que represente cada fila de la tabla, una clase simple con los campos privados que se accedan con los métodos públicos getters y setters. Como los datos de la bbdd pueden ser nulos se evita utilizar tipos de datos primitivos.

Se puede usar el tag h:commandLink o h:commandButton en una o más columnas para llamar a un action de MiBean.java y pasarle el objeto MiData apropiado, esto es más rápido que pasarlo con el tag f:attribute con un ID para obtener el elemento seleccionado directamente de la base de datos. Para conocer la fila seleccionada se usa el binding de h:dataTable para dinámicamente sincronizar el estado de la tabla con el backing bean.

Se muestra un ejemplo:

```
<h:form>
    <h:dataTable
        value=

"#{myBean.myDataList}"


var=

"myDataItem"
        binding=

"#{myBean.myDataTable}">
        <h:column>
            <f:facet name=

"header">
                <h:outputText value=

"ID" />
            </f:facet>
            <h:commandLink action=

"#{myBean.editMyData}">
                <h:outputText value=

"#{myDataItem.id}" />
            </h:commandLink>
        </h:column>
        ...
    </h:dataTable>
</h:form>
```

Obtener un registro seleccionado

En el código anterior se indica que al hacer click en h:commandLink se invocará el método editMyData() de MyBean.

El elemento MyData que corresponde a ese registro puede ser obtenido con el método getRowData() de la clase HtmlDataTable. Extendiendo la clase MiBean con el siguiente código:

```
package ejemplo;

import javax.faces.component.html.HtmlDataTable;

public class MyBean {


private HtmlDataTable myDataTable;


private MyData myDataItem =

new MyData();


public
```

```
String editMyData() {
        // Obtener el elemento MyData para editarlo
        myDataItem = (MyData) getMyDataTable().getRowData();


return

"edit"; // Navega hacia edit
    }



public HtmlDataTable getMyDataTable() {


return myDataTable;
    }


public MyData getMyDataItem() {


return myDataItem;
    }


public void setMyDataTable(HtmlDataTable myDataTable) {


this.myDataTable = myDataTable;
    }


public void setMyDataItem(MyData myDataItem) {


this.myDataItem = myDataItem;
    }
}
```

Seleccionar varios registros utilizando checkboxes. Se pueden seleccionar varios registros, agregando una propiedad de tipo boolean a la clase wrapper en este caso MyDate y en la columna desplegarlo como h:selectBooleanCheckbox.

```
package ejemplo;

public class MyData {


private

boolean selected;


public

boolean isSelected() {


return selected;
    }


public void setSelected(

boolean selected) {
```

```
this.selected = selected;
    }
}
```

Y en la página JSF

```
<h:form>
    <h:dataTable
        value=

"#{myBean.myDataList}"


var=

"myDataItem"
        binding=

"#{myBean.myDataTable}">
        <h:column>
            <f:facet name=

"header">
                <h:outputText value=

"Select" />
            </f:facet>
            <h:selectBooleanCheckbox value=

"#{myDataItem.selected}" />
        </h:column>
        ...
    </h:dataTable>
    <h:commandButton
        action=

"#{myBean.getSelectedItems}"
        value=

"Elementos seleccionados"/>
</h:form>
```

Ciclo de vida del listener

Puede ser útil para realizar un debug y cuando se están dando los primeros pasos con JSF el poner un breakpoint de las diferentes etapas del ciclo de vida de una página JSF, así se puede conocer la cadena de acciones que hace tras bambalinas.
Las 6 fases de la vida JSF son:

1. Restaurar vista
2. Asignar valores de petición
3. Realizar validaciones
4. Actualizar los valores del modelo
5. Invocar la aplicación
6. Presentar las respuestas

Este sería un ejemplo de un LifeCycleListener:

```
package ejemplo;

import javax.faces.event.PhaseEvent;
```

```
import javax.faces.event.PhaseId;

import javax.faces.event.PhaseListener;

public class LifeCycleListener

implements PhaseListener {


public void beforePhase(PhaseEvent event) {


System.out.println(

"Fase Anterior: " + event.getPhaseId());
    }


public void afterPhase(PhaseEvent event) {


System.out.println(

"Fase Posterior: " + event.getPhaseId());
    }


public PhaseId getPhaseId() {


return PhaseId.ANY_PHASE;
    }
}
```

Se agrega al fichero faces-config.xml

```
<lifecycle>
    <phase-listener>mypackage.LifeCycleListener</phase-listener>
</lifecycle>
```

Obteniendo por resultado la siguiente salida

```
Fase Anterior: RESTORE_VIEW 1
Fase Posterior: RESTORE_VIEW 1
Fase Anterior: APPLY_REQUEST_VALUES 2
Fase Posterior: APPLY_REQUEST_VALUES 2
Fase Anterior: PROCESS_VALIDATIONS 3
Fase Posterior: PROCESS_VALIDATIONS 3
Fase Anterior: UPDATE_MODEL_VALUES 4
Fase Posterior: UPDATE_MODEL_VALUES 4
Fase Anterior: INVOKE_APPLICATION 5
Fase Posterior: INVOKE_APPLICATION 5
Fase Anterior: RENDER_RESPONSE 6
Fase Posterior: RENDER_RESPONSE 6
```

Mensajes de error

En JSF, se pueden configurar paquetes de recursos y personalizar los mensajes de error para convertidores y validadores. El paquete de recursos se configura dentro de faces-config.xml:

```
<message-bundle>catalog.view.bundle.Messages</message-bundle>
```

Las parejas clave-valor de los mensajes de error se añaden al fichero Message.properties:

```
#conversion error messages
javax.faces.component.UIInput.CONVERSION=Input data is not in the correct type.
```

```
#validation error messages
javax.faces.component.UIInput.REQUIRED=Required value is missing.
```

Recomendaciones de diseño

Beans

JSF presenta dos nuevos términos:

- *managed bean* (objeto manejado): Un managed bean describe cómo se crea y se maneja un bean. No tiene nada que ver con las funcionalidades del bean.
- *backing bean* (objeto de respaldo). El backing bean define las propiedades y la lógica de manejo asociadas con los componentes UI utilizados en la página. Cada propiedad del bean de respaldo está unida a un ejemplar de un componente o a su valor. Un backing bean también define un conjunto de métodos que realizan funciones para el componente, como validar los datos del componente, manejar los eventos que dispara el componente y realizar el procesamiento asociado con la navegación cuando el componente se activa.

Una típica aplicación JSF acopla un backing bean con cada página de la aplicación. Sin embargo, algunas veces en el mundo real de nuestras aplicaciones, forzar una relación uno-a-uno entre el backing bean y la página no es la solución ideal. Puede causar problemas como la duplicación de código. En el escenario real, varias páginas podrían necesitar compartir el mismo backing bean tras bambalinas.

En comparación con la aproximación ActionForm y Action de Struts, el desarrollo con beans de respaldo de JSF sigue unas mejores prácticas de diseño orientado a objetos. Un backing bean no sólo contiene los datos para ver, también el comportamiento relacionado con esos datos. En Struts, Action y ActionForm contienen los datos y la lógica por separado.

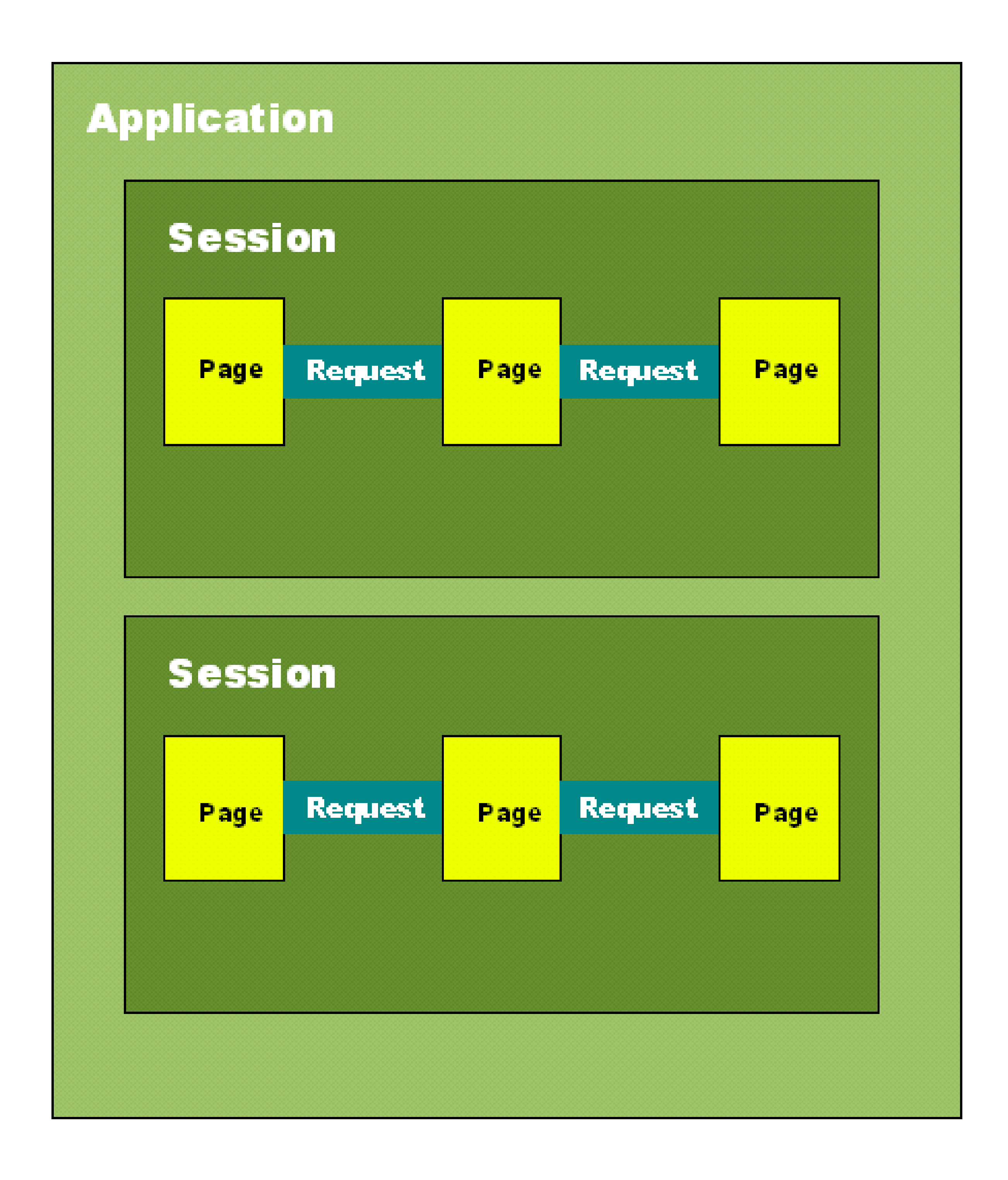
Manejo del scope y los managed beans

Se denomina *scope* a la disponibilidad (o contexto) de un objeto y a su período de *vida* en una aplicación web.

Se verá en un ejemplo de aplicación de encuesta la utilización de un objeto con scope application para guardar los votos, un objeto de scope session para asegurar que el usuario solo pueda votar una vez por sesión, se utilizará también un objeto con scope request para mostrar en pantalla la hora en que el usuario emitió su voto, es de tipo request porque ese valor ya no se necesitará más una vez mostrado.

Tan pronto como un usuario está en una pagina, los valores de los componentes son "recordados" cuando se muestra la página por ejemplo el hecho de que el usuario haga click en un botón que regrese null. Sin embargo cuando abandona la página el valor del componente desaparece.

Para tener disponibles valores en otras páginas o incluso en la misma página lo lógico sería guardar los valores. Antes de crear una propiedad para que guarde un valor, se debe determinar el scope apropiado para ese valor, ya que muchos usuarios podrían acceder a la aplicación al mismo tiempo, así que se necesitaría utilizar el scope más *corto* posible para hacer un mejor uso de los recursos del servidor. La siguiente figura muestra la duración de cada tipo de scope.



Por ejemplo una aplicación que tenga una lista desplegable con tipos de medidas (píxeles, centímetros y pulgadas), se tendría que guardar en un ApplicationBean1, así todos los usuarios en sesiones concurrentes podrían compartir la lista. Por otro lado el nombre del usuario logado se guardaría en un SessionBean1 para que el nombre esté disponible en todas las páginas que el usuario acceda durante su sesión. En caso de no necesitar alguna información más allá de la petición actual entonces se utilizaría el RequestBean1.

Validadores

Los validadores estándar que vienen con JSF son básicos y podrían no cumplir con los requerimientos de la aplicación, pero desarrollar validadores JSF propios es sencillo, aunque lo más recomendable es utilizar los built-in y sólo en casos de extremo detalle o complejidad de la validación crear uno pero extendiendo de la clase base de JSF.

Paginación

Cuando la información a presentar en una página es demasiada, se debe optar por partirla en varios apartados o páginas. Permitiendo desplazarse al usuario entre todas ellas, el típico *Página 1 < Atrás | 1 2 3 4 | Adelante >*.

La paginación debe hacerse desde back-end es decir desde las consultas a la base de datos si la cantidad de registros es muy grande y no sólo a nivel front-end. En el caso de displaytag es recomendable utilizarlo cuando las colecciones de objetos son pequeñas ya que sube a sesión toda la colección para después manipularla.

Buenas prácticas

Estructura y diseño

La estructura de directorios estándar de una aplicación JSF es la siguiente:

```
proyect\MyApp \\
\src \\
->YourBeans.java \\
\web \\
->YourJSP-pages.jsp \\
->AnyImages.gif \\
\WEB-INF \\
->web.xml \\
->faces-config.xml
```

Donde en el directorio ##src## se sitúan los beans de la aplicación, en ##web## todas las páginas jsp ó html, así como las imágenes y demás archivos necesarios para la aplicación, y en ##WEB-INF## los ficheros de configuración como el fichero descriptor de despliegue (web.xml) y el de configuración JSF (faces-config.xml).

Codificación

En cuanto a la importación de librerías, se debe usar el prefijo ##f## para hacer referencia a etiquetas del núcleo de la implementación mientras que el prefijo ##h## para hacer referencia a etiquetas de componentes HTML:

```
<%@ taglib uri=

"http://java.sun.com/jsf/html" prefix=

"h" %>
<%@ taglib uri=

"http://java.sun.com/jsf/core" prefix=

"f" %>
```

Respecto a los validadores empleados, es recomendable la creación de un catálogo de validadores, fácilmente accesible, donde se encontrará la relación entre validadores, objetos de negocio y controles. Estos validadores se crearán en el mismo paquete, al igual que las clases para las etiquetas JSP. El nombre del validador deberá identificar, como mínimo, el control o tipo de control sobre el que actúa y la acción de validación que realiza. Sería recomendable que también el validador incluyera comentarios en el código, tanto para comentar el proceso como para generar documentación javadoc.

Subvistas dinámicas

En cuanto al uso de scriptlets, La etiqueta `jsp:include` sólo acepta el uso de scriptlets. Se requiere que el managed bean tenga el alcance sesión, de lo contrario, surgirán problemas al usar el alcance request. Sin embargo, los mismos problemas surgen al usar múltiples subvistas e incluir etiquetas como se describe arriba.

El código XML relevante del fichero faces-config es el siguiente:

```
<managed-bean>
    <managed-bean-name>myBean</managed-bean-name>
    <managed-bean-class>mypackage.MyBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

El código relevante del fichero JSP principal es:

```
<%
    mypackage.MyBean myBean = (mypackage.MyBean) session.getAttribute(

"myBean");


if (myBean ==

null) { // First-time initialization of bean not done yet.
        myBean =

new mypackage.MyBean();
    }


String includePage = myBean.getIncludePage() +

".jsp";
%>
<f:view>
    <html>
        <head>
        </head>
        <body>
            <h:panelGroup rendered=

"#{myBean.includePage !=

null}"
>
                <jsp:include page=

"<%= includePage %>" />
            </h:panelGroup>
        </body>
    </html>
</f:view>
```

IMPORTANTE: no usar f:subview en lugar de h:panelGroup. Todas las páginas incluidas deberían tener su propio f:subview con un único ID. El código java del backing bean MyBean.java debería parecerse al siguiente:

```
private

String includePage;

public

String getIncludePage() {


if (...) { // Do your thing,

this is just a basic example.
        includePage =

"includePage1";
    }

else

if (...) {
        includePage =

"includePage2";
    }

else

if (...) {
        includePage =

"includePage3";
    }


return includePage;
}
```

Aplicación de ejemplo

Dentro del catálogo interno de la Junta de Andalucía se encuentra el proyecto CRIJA, en el cual se hace uso del framework JSF. Este proyecto se encarga del mantenimiento del censo de equipos microinformáticos. En CRIJA se utilizan páginas xhtml las cuales serán referenciadas mediante extensiones .jsf para ser controladas por el FacesServlet definido en el web.xml:

```
<!-- Configuracion Faclets -->
    <context-param>
        <param-name>facelets.LIBRARIES</param-name>
        <param-value>/WEB-INF/facelet/tomahawk.taglib.xml;/WEB-INF/facelet/viavansi-custom.taglib.xml</param-value>
    </context-param>
    <context-param>
        <param-name>facelets.DEVELOPMENT</param-name>
        <param-value>

true</param-value>
    </context-param>
    <!-- Solo durante el desarrollo, en produccion quitar-->
    <context-param>
        <param-name>facelets.REFRESH_PERIOD</param-name>
        <param-value>1</param-value>
    </context-param>
    <!-- Los comentarios xhtml son ignorados -->
    <context-param>
```

```
        <param-name>facelets.SKIP_COMMENTS</param-name>
        <param-value>

true</param-value>
    </context-param>

        <!-- Configuracion JSF-->
    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>server</param-value>
    </context-param>
        <context-param>
                <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
                <param-value>.xhtml</param-value>
        </context-param>
        <!--<context-param>
                <param-name>org.apache.myfaces.PRETTY_HTML</param-name>
                <param-value>

true</param-value>
        </context-param>-->
        <context-param>
                <param-name>org.apache.myfaces.ALLOW_JAVASCRIPT</param-name>
                <param-value>

false</param-value>
        </context-param>
        <context-param>
                <param-name>javax.faces.CONFIG_FILES</param-name>
                <param-value>
                        /WEB-INF/faces-config.xml,
                        /WEB-INF/admin-faces-config.xml
                </param-value>
                <description>
                        Ficheros de definicion
                </description>
        </context-param>
    <context-param>
        <param-name>com.sun.faces.validateXml</param-name>
        <param-value>

true</param-value>
        <description>
            Set

this flag to

true

if you want the JavaServer Faces
            Reference Implementation to validate the XML in your
            faces-config.xml resources against the DTD.  Default
            value is

false.
        </description>
    </context-param>
    <context-param>
        <param-name>com.sun.faces.verifyObjects</param-name>
        <param-value>

true</param-value>
        <description>
            Set

this flag to

true

if you want the JavaServer Faces
            Reference Implementation to verify that all of the application
            objects you have configured (components, converters,
```

```
            renderers, and validators) can be successfully created.
            Default value is

false.
        </description>
    </context-param>
```

Una vez que se configura el fichero web.xml las páginas xhtml se desarrollarán atendiendo a a las librerías incorporadas:

```
<html xmlns=

"http://www.w3.org/1999/xhtml"
                xmlns:ui=

"http://java.sun.com/jsf/facelets"
                xmlns:h=

"http://java.sun.com/jsf/html"
                xmlns:f=

"http://java.sun.com/jsf/core"
                xmlns:v=

"http://www.viavansi.com/jsf/custom"
                xmlns:t=

"http://myfaces.apache.org/tomahawk"
                xmlns:a4j=

"https://ajax4jsf.dev.java.net/ajax"
                xmlns:c=

"http://java.sun.com/jstl/core">
```

En este caso extraído de la página index.xhtml del directorio web /admin/actualización se hace referencia a las librerías de tomahawk y myFaces que serán utilizadas a través de etiquetas JSTL. Además, se utilizan tags propios definidos por el usuario a través de la etiqueta 'v:' .De esta manera, la página quedaría estructurada como sigue:

```
<ui:composition template=

"/includes/${skin}/template.xhtml">
        <ui:define name=

"title">Actualizaci##³n Masiva de Modelos</ui:define>
        <ui:define name=

"head">
                <v:script js=

"embedded, jquery.highlightFade, forms.help,                                      search.loading, tables" />
        </ui:define>
        <ui:define name=

"header">
                <h2>Actualizaci##³n Masiva de Modelos</h2>
        </ui:define>
        <ui:define name=

"body">
        <!-- enctype=

"multipart/form-data" -->
                <v:form id=

"formularioModelos">
                        <t:htmlTag value=
```

```
"div" styleClass=

"mensajes error"                                          rendered=

"#{not empty requestScope.resultadoUpdate}">
                                <ul><li>
                                        <span class=

"error">#{resultadoUpdate}</span>
                                </li></ul>
                        </t:htmlTag>
                        <p>
                                Este proceso actualizar##¡ el modelo de todos los
                                Posteriormente, eliminar##¡ el modelo original.
                                <br/>
                                La principal utilidad consiste en fusionar modelos
                        </p>
                        <v:inputText label=

"ID Modelo Original (se borrar##¡)"                                 value=

"#{adminActualizaModelosController.idmodeloOrigen}"
                                title=

"ID MODELO ORIGEN" id=

"idmodeloOrigen"
                                size=

"10"
                                required =

"

true"

                                maxlength=

"10"/>

                        <v:inputText label=

"ID Modelo Destino (todos los modelos                               tendr##¡n este ID Modelo)"

"#{adminActualizaModelosController.idmodeloDestino}"
                                title=

"ID MODELO DESTINO" id=

"idmodeloDestino"
                                size=

"10"
                                required =

"

true"

                                maxlength=

"10"/>

             <p class=

"botonera">
                <h:commandButton styleClass=

"boton"
                        value=

"Enviar"                                                                      action=
```

```
"#{adminActualizaModelosController.actionProcesaModelos}" />                </p>
        </v:form>
                <!-- Volver a la p##¡gina anterior -->
                <v:volverPanel label=

"${msg.volver}"                                                              title=

"${msg.volverApanelDeControl}">
                </v:volverPanel>
  </ui:define>
</ui:composition>
```